Empirical Study of Parallel LRU Simulation Algorithms

David M. Nicol [*]
College of William and Mary
Williamsburg, VA 23187-9795

Eric Carr
Carleton College
Northfield, MN 55057

Abstract

This paper reports on the performance of five parallel algorithms for simulating a fully associative cache operating under the LRU (Least-Recently-Used) replacement policy. Three of the algorithms are SIMD, and are implemented on the MasPar MP-2 architecture. Two other algorithms are parallelizations of an efficient serial algorithm on the Intel Paragon. One SIMD algorithm is quite simple, but its cost is linear in the cache size. The two other SIMD algorithm are more complex, but have costs that are independent of the cache size. Both the second and third SIMD algorithms compute all stack distances; the second SIMD algorithm is completely general, whereas the third SIMD algorithm presumes and takes advantage of bounds on the range of reference tags. Both MIMD algorithm implemented on the Paragon are general, and compute all stack distances; they differ in one step that may affect their respective scalability. We assess the strengths and weaknesses of these algorithms as a function of problem size and characteristics, and compare their performance on traces derived from execution of three SPEC benchmark programs.

[*]This work was partially supported by the National Aeronautics and Space Administration under NASA contract number NASA1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA, 23681. It was also supported in part by NSF Grant CCR 9201195.

1 Introduction

Associative caches arise in many contexts of computer systems: construction of memory hierarchies is a notable case, caches are used also in file systems and databases. Any realizable cache has finite size, and periodically one element must be discarded to make room for another. The Least-Recently-Used (LRU) policy is frequently employed to select the element to be discarded, i.e., the item discarded is the one whose last access time is farthest in the past.

A trace-driven cache simulation accepts as input a reference string $x_1, x_2, \ldots, x_N$ of N symbols. Each symbol identifies some cacheable element of memory, e.g., a cache line. Serial cache simulations process the reference string one reference symbol at a time, with each new reference the simulation updates internal data structures. For this reason it is convenient to describe time in terms of reference index—reference x_t is presented (but not yet processed) at time t. Ultimately one asks the simulation to determine for a given cache size C (and frequently other parameters, such as line length) the fraction of references x_t such that the symbol referenced at time t is a "hit", i.e., is already present in the cache. System designers wish to make caches as small as possible while still achieving high hit ratios. Since referencing behavior largely determines the cache's performance, designers customarily drive cache simulations with very long traces of observed references. The LRU policy enjoys the "stack property" [5], which asserts that for any given trace, at any time t a cache with capacity $C + 1$ will contain all the references that a cache with capacity C would hold at time t. The stack property allows one, for a given trace, to compute the hit ratio of a cache of any size by knowledge of the *stack distances*. The stack distance of the t^{th} reference is the smallest sized cache that at time t already contains this reference's symbol. To determine the hit ratio for a cache of size C, one finds the fraction of references whose stack distance is C or smaller.

Parallel algorithms for determining hit ratios for given traces are described in [3], and [6]. Only [6] reports actual performance data, and that only on randomly generated traces. The contribution of the present paper is to report on implementations of variants of these algorithms, in order to provide a better understanding of the tradeoffs inherent in choosing an architecture, and algorithm, for parallel cache simulation.

Three of the algorithms upon which we report are SIMD algorithms, and rely upon massively parallel operations such as scans [2], and sorting. The first is the "level-by-level" algorithm described in [6]. This algorithm computes the stack distances for all references whose stack distances are C or smaller, C being the maximum number of references the cache holds. The performance results previously reported were from the MasPar MP-1 architecture, based on a randomly generated trace; the results reported here are from the MasPar MP-2, both on random traces and traces of SPEC92 benchmark program executions [9]. The algorithm has a computational cost that is linear in the maximal cache size considered, C. In our study we consider PEs saturated at 1024 references each, and execute on 1K, 4K, and 16K PE machines. For each machine size we vary C to determine the base cost and sensitivity to this linear term. The SPEC traces are executed only on the 1K PE machine.

The second and third algorithms are variants of the "geometric" algorithm described in [6]. We present the first known performance results for this algorithm. The algorithms differ in how they determine for each reference the identity of the closest next reference in the reference string. One implementation is completely insensitive to the distribution of symbols in the reference string; instead we consider its sensitivity to increasing reference string length, as well as its performance on SPEC traces. The other algorithm is similar, except it presumes and exploits bounds on the range B of reference symbols. When the reference symbols can be bounded by a relatively small range, this algorithm is considerably faster the the general algorithm. Our study of this algorithm concentrates then on determining the sensitivity of performance to increasing B.

The MIMD algorithms studied are parallelizations of the efficient serial LRU simulation algorithm described in [7] and [10]. This algorithm works in the following manner. For every reference x_t, let $n(t)$ be the smallest index $u > t$ such that $x_t = x_u$, and let $p(t)$ be the largest index $v < t$ such that $x_t = x_v$. We call $n(t)$ and $p(t)$ the "next" and "previous" occurrences of symbol x_t. The serial algorithm we use (as well as the parallel geometric algorithms) exploit the fact that the stack distance of x_t is equal to the number of unique symbols referenced between $p(t)$ and x_t, plus one. Given reference x_t, one hashes on x_t's symbol to find the reference index of the last time that symbol was accessed. A second data structure, a search tree, organizes all unique symbols seen so far, based the last reference time. Finding x_t's representation in the tree (with search key $p(t)$), we count the number of symbols in the tree with larger index. All of this is accomplished with average cost that is logarithmic in the stack distance. We parallelized this algorithm using the idea of [3] to split the reference trace temporally, and employ a "fix-up" phase at the end. Our parallelized version was implemented on the Intel Paragon, using up to 64 processors. Our objective in this study is to determine speedups achievable simulating moderately long SPEC references (8M to 95M references each). Of particular interest is the relative cost of the fix-up phase, and of two different ways of performing the fix-up. One method uses as many communication steps as there are processors, the other uses only a logarithmic number of communication steps. Surprisingly (and for reasons we later identify) the latter approach is considerably less efficient on the traces we consider.

The remainder of the paper is organized as follows. Section 2 concerns the level-by-level algorithm; section 3 describes the two geometric LRU algorithms. Both of these sections report on performance observed using 1K, 4K, and 16K Maspar MP-2's, on randomly generated reference strings. Section 4 describes the MIMD implementation of the efficient serial algorithm; comparative performance of SIMD and MIMD algorithms on traces of three SPEC92 benchmark programs is examined in Section 5. Section 6 presents our conclusions.

2 Level By Level Algorithm

One reason stack algorithms are so named is that one can always order (i.e., stack) all references in a cache by the stack distance each would have if the next reference symbol were to name it. The seminal paper on stack policies is [5]. The position of a symbol b in the stack at reference t is precisely the stack distance of x_t if $x_t = b$. We may then view the behavior of a serial cache simulation algorithm in terms of how the stack evolves as each reference is processed. Visually, the effect of processing x_t is to remove x_t from the stack, "push" all symbols above that position down one slot and place x_t at the top of the stack. Thus, a symbol b's relative position in the stack is unaffected at time t if the symbol x_t has a smaller stack distance than would b, if $x_t = b$. As a consequence, if reference b attains position j in the stack at time t, it will remain so up to some time z where either $x_z = b$, or x_z references some symbol that is further down the stack than b. This is illustrated in Figure 1, where arrows depict the migration of symbols downward in the LRU stack until the symbols are re-referenced. The progression of symbol b is highlighted. Observe that symbol b remains at the same level j in the cache over the longest contiguous sequence $x_t, x_{t+1}, \ldots, x_{z-1}$ such that each reference in the sequence has a stack distance of $j-1$ or smaller—each is a hit in a cache of size $j-1$. This observation is key, for if we can mark each reference in the string as being a hit or miss in a cache of size $j-1$, the references that are marked as misses also mark precisely where new symbols attain stack level j. *The symbols at stack level j do not change at any reference that is a hit in a cache of size $j-1$.* Using this observation, given the hit/miss status of every reference in a cache of size $j-1$, we can determine the symbols at stack level j at every point in time by marking each position where there is a miss, noting which symbol attains level j at that point, and copy that symbol across the largest possible contiguous sequence of hits. New hit/miss markings are computed following this copy step.

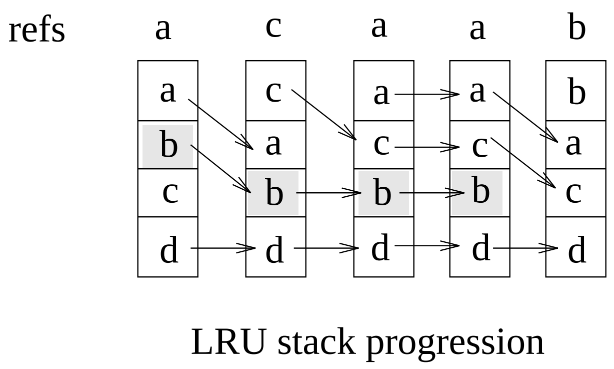

Figure 1: Modification of LRU stack

PEs	trace	$C = 4$	$C = 16$	$C = 256$	$C = 1024$
1K	rand-4	0.13	0.64	10.7	43.0
1K	rand-1024	0.13	0.58	9.6	38.8
4K	rand-4	3.3e-2	0.14	2.71	10.8
4K	rand-1024	3.3e-2	0.14	2.40	9.7
16K	rand-4	8.4e-3	4.0e-3	0.67	2.7
16K	rand-1024	8.4e-3	3.6e-3	0.59	2.4

Table 1: Wallclock processing time per reference, in microseconds, for the level-by-level algorithm, as a function of cache size C

The copy step is accomplished in parallel using a segmented copy-scan[2]. Techniques like this are standard in SIMD programming, indeed the MasPar library contains numerous variations on scan operations. The level-by-level algorithm computes stack distances a level at a time. First, all references with stack distance 1 are determined. Next, all references with stack distance 2 are determined, and so on. The complexity of each step is $O(N/P + \log P)$ on an EREW machine, the MasPar's asymptotic complexity is slightly higher owing to its use of a mesh interconnection network.

The level-by-level algorithm considers the entire reference trace to be spread across all PEs, the first PE receiving the first N/P references, the second PE receiving the second N/P references, and so on. An advantage of this method is its simplicity, the entire program is barely 150 lines of code long. The major disadvantage is that the execution time is linear in the number of stack levels considered, which makes it ill-suited for contexts where the stack distance of every reference is required (not just those within the maximum cache size).

The experimental results in Table 1 measure the sensitivity of the algorithm to the maximum cache size C, and show that the algorithm scales—that performance remains good as the problem size and architecture size are simultaneously increased. The performance shown is based on strings where each element is sampled uniformly at random from $[1, k]$ (rand-k), with $k = 4$ and $k = 1024$. We provide data from runs on MasPar MP-2 machines with 1K, 4K, and 16K PEs; in all cases each PE is responsible for 1024 references. The performance figure given is the wallclock time per reference (in microseconds) expended simulating the entire trace (i.e., the parallel execution time divided by the total number of references). The most important things to note about this table are

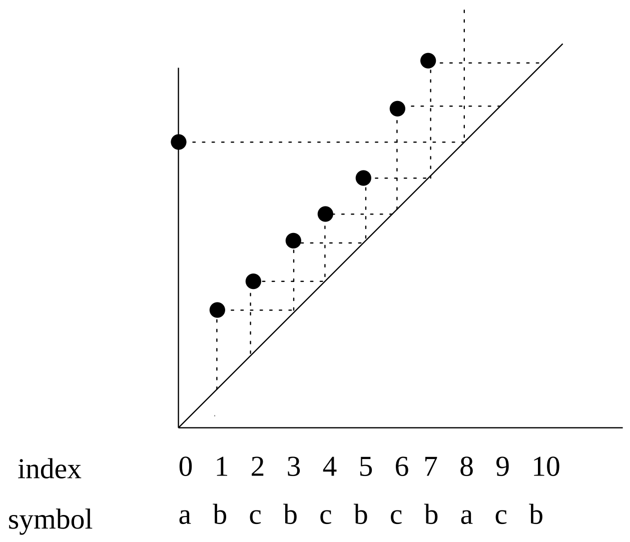

Figure 2: Geometric interpretation of LRU cache behavior

- for small cache sizes the algorithm is *fast*. This will be even more apparent when we see that our implementation of the efficient serial algorithm, uses over 30 microseconds per reference on a 60MHz Sparc-20,
- performance degrades linearly, almost perfectly, as C increases,
- performance improves linearly, almost perfectly, as the number of processors increases.
- There is a slight sensitivity to the number of symbols in the reference string. This is actually a sensitivity to the hit ratio, as the cost of performing segmented scans increases as the number of segments decrease, i.e., as the hit ratio increases..

The scalability and essential speed of the algorithm show that it should be seriously considered for simulation of small sets, such as those commonly used in set-associative caches in computer systems.

3 Geometric LRU Algorithms

The fundamental idea behind the geometric LRU algorithm is that the number of unique symbols between reference x_t and $p(t)$ can be computed from a geometric analysis of a two-dimensional plane wherein are are plotted all points of the form $(t, n(t))$. Figure 2 illustrates the point by plotting all points $(t, n(t))$ for a sample reference string, and connects points associated with a common symbol with line segments. Between the reference to symbol a at $t = 0$ and $t = 8$ we find seven references—but only two unique symbols. In order to count each unique symbol only once, for each symbol occurring in $[t, n(t)]$ we identify the last reference to that symbol by detecting references $u \in [t + 1, n(t) - 1]$ such that $n(u) > n(t)$. In Figure 2 we see two of these.

To count these symbols in parallel we use the notion of a point (x, y)'s *rank*–the total number of plotted points (u, v) that *dominate* (x, y), i.e., $x < u$ and $y < v$. The stack distance of x_t is one plus the number of points points of the form $(u, n(u))$ with $p(t) < u < t$ that dominate $(p(t), t)$. Since every point $(v, n(v))$ with $v > t$ dominates $(p(t), t)$, we find the stack distance of x_t by computing the rank of $(p(t), t)$, and subtract

$N-t$. (Note that this is a simplification of the calculation given in [6]). The rank of $(0,8)$ in Figure 2 is two, hence the stack distance of x_8 is three.

Ranks can be computed on a SIMD machine by parallelizing an algorithm based on the multidimensional divide-and-conquer paradigm described in [1], as follows. Let us say that the rank of a point (x,y) *over interval* $[u,v]$ is the total number of plotted points of the form (a,b) that dominate (x,y), and $x<a<u$. Then for any $z>v$, the rank of (x,y) over $[u,z]$ is equal to the rank of (x,y) over $[u,v]$, plus the number of plotted points whose X coordinate is in $[v+1,z]$ and whose Y coordinate exceeds y (note we assume integer coordinate values throughout). For any $c\in[v+1,z]$ the rank of (c,d) over $[u,z]$ is just its rank over $[v,z]$.

Ranks can be computed in parallel using parallel merge and parallel prefix operations, as follows. Imagine that we have two equal length adjacent subsequences of references (say, 0-3 and 4-7) sorted on their $n(t)$ values. References x_t for which $n(t)$ is not defined (i.e., the last reference in the string to a particular symbol) are taken to have $n(t)=N+t$, but will not have meaningful ranks assigned. Presume further that the rank of each reference over its present subsequence is known. We associate a tag of 0 with every reference in the "left" subsequence, and a tag of 1 with every reference in the "right" subsequence. A partial trace, and partitioning into subsequences 0-3 and 4-7 is illustrated in Figure 3. Next we merge the two subsequences on the $n(t)$ value into a larger sequence, carrying along in the merge the t, rank, and tag fields. This merge step can be performed in parallel, e.g., using a bitonic merge (see [4]). Since this merge executes precisely the same number of instructions regardless of symbol value, the algorithm based on it has performance that is independent of the symbol string. Following the merge we update the ranks. Any reference position whose tag field is 0 adds to its rank field the number of set tags that lie to its right in the new sequence. This count is precisely the number of points from the right subsequence whose $n(t)$ value dominates the reference. Now these counts can be obtained in parallel for all references using a "postfix-sum" operation (a prefix-sum running from right to left). The final step is to prepare the new sequence for the next step by determining (by position) whether the sequence is "left" or "right" at the next level of the algorithm, and to set the tag bits accordingly.

Using this basic operation, the stack distances are computed in $\log N$ steps. Given that the $n(t)$ values are known, the first step partitions the reference string into N subsequences, each of length 1. Each group is trivially sorted on the $n(t)$ value, and the ranks are all trivially zero. The merge/postfix step is carried out on all groups in parallel, creating subsequences of length 2, which are merged/postfixed into subsequences of length 4, and so on, until the last merge/postfix step leaves us with the entire reference string as a single sequence. The record associated with x_t holds the rank of $n(t)$ (if defined), and hence the stack distance of $n(t)$. A final assignment moving the stack distance of $n(t)$ to its own reference and processor may be performed, although this is not logically necessary for the computation of hit ratios.

Both of the geometric algorithms we study use the merge/postfix operations above. They differ in their computation of the $n(t)$ values. The general algorithm computes the $n(t)$ values by first sorting stably on the reference symbol. Two-word records are carried along in the sort, the symbol as well as the reference index where the symbol occurred. Following this step the $n(t)$ and $p(t)$ values of reference x_t are easily discovered by examining the reference index field of adjacent records. The $n(t)$ and $p(t)$ values can be carried back to x_t's "home" location with another sort, this one on t. Our implementation of the sort is based on Jan Prins' implementation of the bitonic sort algorithm [8]. This approach is convenient, but is overkill in the sense that the total sorted order is not required to find $n(t)$ and $p(t)$. As we will see, the convenience comes at a severe performance cost.

The second geometric algorithm assumes that all reference symbols lie in a range $[0,B]$. Then, for each symbol $b\in[0,B]$, a backwards segmented copy-scan is performed where segments begin with references whose symbol is b, and the value propagated (backwards) is the symbol's reference index. Following the

n(t)	1	5	2	4	7	6	8	-	-	-
t	0	1	2	3	4	5	6	7	8	9
symbol	a	a	b	c	c	a	a	c	a	b

Partial trace

n(t)	1	4	5	9	6	7	8	N+7
t	0	3	1	2	5	4	6	7
rank(n(t))	3	0	1	0	2	2	1	-
tag	0	0	0	0	1	1	1	1

Pre-merge state

n(t)	1	4	5	6	7	8	9	N+7
t	0	3	1	5	4	6	2	7
rank(n(t))	7	4	5	2	2	1	1	-
tag	0	0	0	1	1	1	0	1

Post merge/postfix state

Figure 3: Parallel computation of ranks using merge and postfix operations

copy-scan, the reference immediately to the right of one with symbol b will hold the reference position of the nearest reference to the right with the same symbol. The copy-scans are fast, but the simulation's cost degrades linearly in B.

Table 2 plots the wallclock processing time per reference of the general geometric algorithm, as a function of the number of references mapped to each PE, and the architecture used. The most striking feature of this data relates to this algorithm's ability to scale. There is clearly some advantage to increasing the number of references assigned to a PE, up to a point, after this the performance begins to decline. The decline is even more pronounced as we multiply the number of PE's used—the 16K PE instance of the problem runs only 6 times faster than the 1K version. However, consider that for a fixed number of references/PE the 16K PE problem processes a reference string that is 16 times longer than the corresponding 1K PE problem and which contains the entire trace processed in the 1K PE case. The 16K PE solution determines stack distances for some references in the 1K PE substring that the 1K PE solution did not (references x_t where $n(t)$ lies outside of the substring). To compare the algorithms on a completely fair basis we would need to modify the algorithm to permit the 1K machine solve exactly the same problem as does the 16K machine. However, just an adjustment would not matter a great deal on the problems reported here, as its additional overhead is quite small. One approaches the problem with phases, in each phase the reference string processed consists of references x_t from previous phases whose $n(t)$ not yet been discovered, and new references. At the end of a phase all references x_t without matching $n(t)$ are together at the right end of the machine (being sorted on their $n(t)$ values, defined, conveniently, to push them to the right). To prepare for the next phase we simply need to move those references to the left end of the machine. On the runs presented here the number of unique symbols is 1024 or 4, meaning that the amount of data to be moved is not large, and that the number of symbols left over from previous phases is insignificant compared to the number of new references brought in for a new phase. To explain the failure to scale we must look to a different source. The cause is understood when we examine the bitonic sort underlying the algorithm. Its cost increases in the *square*

	References per PE									
PEs	2	4	8	16	32	64	128	256	512	1024
1K	98.8	69.5	27.0	23.8	22.7	22.7	22.7	23.2	23.8	24.5
4K	16.5	12.4	10.5	9.69	9.38	9.33	9.41	9.55	9.73	9.92
16K	5.93	4.88	4.33	4.11	4.03	4.02	4.04	4.09	4.13	4.19

Table 2: Wallclock processing time per reference, in microseconds, for the general geometric LRU algorithm, as a function of the number of references/PE and number of PEs

	Reference range size B												
PEs	4	8	16	32	64	128	256	512	1024	2048	40096	8192	16384
1K	2.63	2.83	3.06	3.42	4.05	5.18	7.35	12.2	17.1	26.3	42.5	73.5	134.8
4K	0.659	0.712	0.771	0.862	1.02	1.30	1.84	2.95	5.15	9.05	14.9	24.2	40.3
16K	0.166	0.178	0.193	0.217	0.257	0.326	0.463	0.740	1.29	2.40	4.60	8.50	14.4

Table 3: Wallclock processing time per reference, in microseconds, for the bounded reference range geometric algorithm, as a function of reference range B and number of PEs

of the logarithm of N, furthermore, the additional steps are more costly as they involve data movement over farther distances (the algorithm uses the MasPar's xnet pipelines). It would appear that both factors contribute significantly to the observed performance, since the ratio $\log^2 2^N / \log^2 2^{N+4}$ accounts for only about 50% of the observed performance degradation.

Now consider the behavior of the version which exploits bounds on the range of symbols. For the data presented in Table 3 we saturate every PE with 1024 references, and consider how performance varies with increasing bound range, B.

Comparison of Table 2 and the 1024 refs/PE data in Table 3 reveals some interesting features. First, that when B is very small, the bounded range algorithm is an order of magnitude faster than the general algorithm. This fact shows that the cost of the general algorithm's implementation is dominated by the $n(t)$ calculation. Second, that for the low values of B, the performance gain is nearly linear as the number of PEs increases. This suggests that the performance degradation observed in the general algorithm's implementation is due to behavior of the $n(t)$ calculation, and not the merge/postfix step. For very large values of B the performance of the bounded reference algorithm scales sublinearly, but is dominated by the cost of repeated segmented copy-scans in the $n(t)$ calculation. The cost of this operation is $O(N/P + P^{1/2})$ on the MasPar, and the degradation may be understood as a result of the $P^{1/2}$ communication term.

Comparison of the two geometric-based algorithms illustrates the importance of the $n(t)$ calculation. There is clearly room for improvement in the general algorithm's implementation; the performance of the bounded reference algorithm with small B gives an upper bound on the improvement we can hope for. It is pointed out in [6] that direct algorithms for computing $n(t)$ in $O(\log N)$ time are known; it appears to be an approach worth investigating.

4 MIMD Algorithms

The fundamental idea for using MIMD computers for trace-driven LRU simulation was reported in [3]—given P processors, N references, and a C line cache, divide the trace into N/P contiguous subsequences, allocated one per processor. *Have each processor assume it begins with an empty cache, but take note of the misses that occur before the cache is full.* At the end of processing its subtrace, processor i sends to processor $i+1$ the contents of its C-line cache at the end of its simulation. These contents are precisely the starting state for processor $i+1$'s simulation, and can be used to resolve the actual hit/miss status of the references processor $i+1$ noted as misses to an non-full cache. If processor $i+1$'s cache is not full at the end of its simulation, it may happen that a symbol reported to it by processor i is in $i+1$'s cache throughout the entire subtrace, and so must be reported to processor $i+2$. This sort of logic carried out in full shows that in the worst case an $O(P)$ communication step is needed to "fix-up" the simulation.

It is convenient in the discussion to follow to consider the processors to be in a linear array, arranged from left to right by increasing index. Our simulation has the processor to which $p(t)$ is assigned compute the stack distance for x_t. After simulating their subtraces, processor i will have references x_s for which $n(s)$ lies on some processor to the right (if at all), and processor $i+1$ will have references x_t for which $p(t)$ lies on some processor to its left (if at all). Consequently, in order to fix-up the simulation between processor i and $i+1$, we may have processor $i+1$ send to processor i a message reporting the identity and reference index of the first occurrence of every unique symbol observed by processor $i+1$. This list, if sorted by increasing reference index, may be treated exactly as additional references to be simulated. However, any new symbol received that processor i did not itself reference must be passed along to processor $i-1$. This observation led us to pipeline the fix-up operation. A processor receives a bundle of unresolved symbols (sorted by increasing reference index) and attempts to resolve each with a match in its own hash table. Failure to find a match causes the symbol and its reference to be placed in a table for later transmission, otherwise the stack distance for the symbol is computed and the symbol is stored in the processor's tree of unique references (this is required to correctly compute the stack distances of references that might later be received during fix-up). At the end of this filtering step the processor sends the accumulated table of still unresolved references (which are still sorted on their reference index) to the processor on its left. The communication complexity of the pipelined approach is $O(P-1)$, since processor 0 receives up to $P-1$ messages.

It has been noted by others[1] that the time-partitioning approach can be extended to simulate many cache sizes and set-associativities, and that the fix-up step can be done with a "fat-tree" merge. The solution described privately to us presumes the existence of many sets; we have adapted however the basic observation that the fix-up process is associative, as follows. Like the pipelined approach, our messages will flow from right to left, and but will be based on postfix computations. The algorithm has $\log P$ steps. Prior to the first step every processor i defines list L_i to include all unique symbols referenced by processor i, sorted by increasing index of first reference time. These references are ones which are unresolved, their stack distances aren't yet known. In each of $\log P$ steps, processor i will send L_i to processor $i-2^j$ provided $i-2^j \geq 0$; it will receive list L_{i+2^j} from processor $i+2^j$, provided $i+2^j \leq P-1$. Processor i filters L_{i+2^j}, checking each reference for a match in processor i's hash table. Any reference not already found in the hash table is appended to L_i, and is inserted in both the hash table and search tree. If a reference x_t is found to already be in the hash table, we check whether the last recorded reference to it was by a reference originally assigned to processor i. If so, its stack distance is computed.

To establish the correctness of the algorithm consider how an unresolved reference x_t originating at processor i propagates. In the first step it is sent to processor $i-1$. Supposing it is not resolved there, in

[1]Private communication from Mark Hill, Gurindar Sohi, and Madhusudhan Talluri

		Number of Processors				
Trace	fix-up	4	8	16	32	64
spec001	pipelined	1.1 (0.7%)	1.3 (1.7%)	1.5 (4.0%)	1.8 (9.1%)	1.9 (17.2%)
spec001	postfix	2.5 (1.6%)	3.0 (3.9%)	4.3 (10.6%)	4.9 (21.9%)	5.2 (36.7%)
spec026	pipelined	2.3 (1.9%)	3.0 (4.8%)	2.0 (6.1%)	1.8 (10.3%)	1.8 (17.4%)
spec026	postfix	4.6 (3.7%)	5.7 (8.7%)	6.4 (17.2%)	6.8 (30.0%)	7.0 (45.2%)
spec090	pipelined	1.1 (0.8%)	1.2 (1.9%)	2.0 (6.0%5)	2.1 (11.4%)	2.2 (20.5%)
spec090	postfix	3.1 (2.4%)	4.1 (6.1%)	5.4 (14.5%)	6.6 (28.4%)	7.5 (46.4%)

Table 4: Fix-up times on three SPEC traces, in seconds, as a function of fix-up method and number of processors. Percentage of total execution time spent in fix-up phase is also given

the second step both i and $i-1$ send it, to processors $i-2$ and $i-3$ respectively. Failing to be resolved at either of these, four processors now send it as part of their lists, to include processors $i-4$,$i-5$,$i-6$, and $i-7$. So long as x_t remains unresolved, at the end of the j^{th} communication step ($j = 0, 1, \ldots, \log P - 1$), all processors $i - 2^j$ to i have x_t in their unresolved lists. Now consider what happens x_t is resolved in step j, e.g., $p(t)$ is found in some processor k between processors $i - 2^j$ and $i - 2^{j-1}$-1. Of course, processor k will not append x_t to L_k. But what of the other processors whose lists already contain x_t? They continue to include x_t in their lists, but for all subsequent communication steps, *x_t will be filtered out by every processor that receives it.* The simple reason for this fact is that any processor to which processors i through $i - 2^j$ send messages will have at that point already received a reference with the same symbol as x_t, because $p(t)$ (or one of its predecessors with the same symbol) is closer to that processor than is any copy of x_t.

The decreased number of communication steps in this approach is counterbalanced by an increased volume of communication and computation. In the pipelined approach an unresolved reference is represented in at most one message at any time. Furthermore, once the reference is resolved no further computation is expended on its behalf. This is not the case with the postfix approach, since every replica of an unresolved reference must be sought for in every processor to which it is sent. There is a trade-off then between the number of parallel communication steps, the communication volume, and the fix-up computation costs. Surprisingly it turns out that on the size of machine we used, up to 64 processors, the pipelined approach enjoyed substantially better performance. Table 4 reports the fix-up times observed using the pipelined and postfix approaches, on 4,8,16,32, and 64 Paragon nodes, on three SPEC benchmark traces (spec001.cexp.pdt, spec026.comp.pdt, spec090.hydro.pdt in the TraceBase depository at New Mexico State University). Each run simulated the first 2^{23} references of the trace. The times shown are in seconds. With each timing we indicate in parenthesis the fraction of the total execution time spent in the fix-up phase.

On the traces studied it is apparent that the additional costs of the scalable fix-up method substantially outweigh its advantages, at least for the range of processors considered. However, one must also bear in mind that when a long subtrace is loaded on each processor (as with the 4 processor data), the fix-up time is only a small fraction of the overall running time.

5 Performance on SPEC Traces

Finally, we present measurements taken from runs driven by traces derived from SPEC92 programs [9]. The traces were obtained from the TraceBase facility, maintained by the New Mexico State University (available

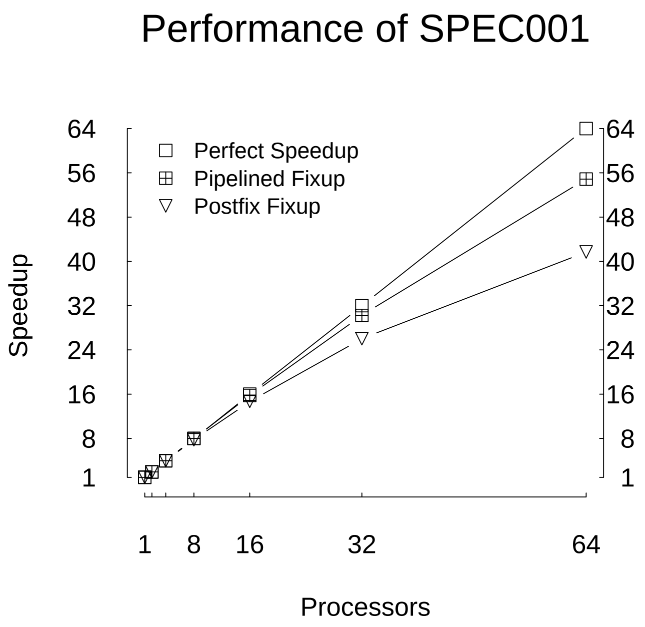

Figure 4: Speedups on `spec001`, for both pipelined and postfix fix-up

by anonymous ftp to tracebase.nmsu.edu). The particular traces used in this study were of the 001, 026, 090, and 097 programs in the SPEC92 suite. Their names in the TraceBase facility are `spec001.cexp.pdt`, `spec026.comp.pdt`, `spec090.hydro.pdt`, and `spec097.nasa7.pdt`. These particular traces were selected for reasons of length. The references were all processed in the order represented in the trace, in particular no distinction is made between data and instruction references. We *do* assume a 16-byte cache line, and so mask off the bottom four bits of every reference.

Figures 4, 5, and 6 plot the speedups of three 8M reference traces on 1 to 64 nodes of the Intel Paragon. For these ratios, the base serial processor timings are 70.8, 59.4, and 61.7 μ-sec/reference, for traces `spec001.cexp.pdt`, `spec026.comp.pdt`, and `spec090.hydro.pdt`, respectively. The simulation is written in C++, and compiled under g++. The perfect speedup line is also plotted on each curve. In considering this data one should keep in mind that the trace length is kept fixed—the individual subtraces are unrealistically small for large processor populations (e.g., 0.25M references/processor for the 64 processor case). Each Paragon processor has approximately 24 Mb available for program and data, given a tremendously long trace one would use all that memory. Nevertheless, the data does make the point that excellent performance is achievable, even when the trace is relatively short.

We did also simulate longer reference strings. `spec097.nasa7.pdt` contains approximately 95 million references. Using 64 processors and varying the string length between 8M and 95M references we observed that the wallclock processing time per reference dropped from 1.34 μ-secs/reference, to 0.89 μ-secs/reference (using pipelined fix-up). This is a 50% improvement due simply to increasing the length of subtrace simulated at each processor. Observe that this level of improvement would bring the 64-processor speedups of the other traces up to near perfection.

Table 5 provides the wallclock processing time per reference, in microseconds, for selected numbers of Paragon processors (using pipelined fix-up), the level-by-level algorithm, the general geometric algorithm,

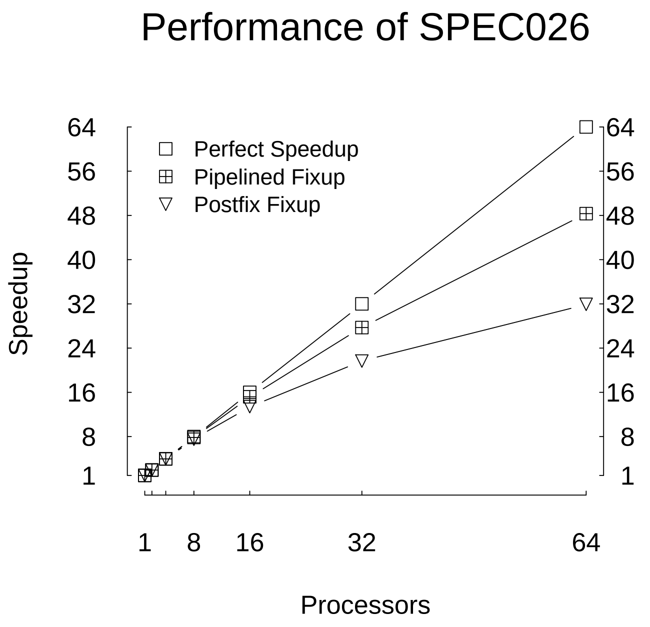

Figure 5: Speedups on `spec026`, for both pipelined and postfix fix-up

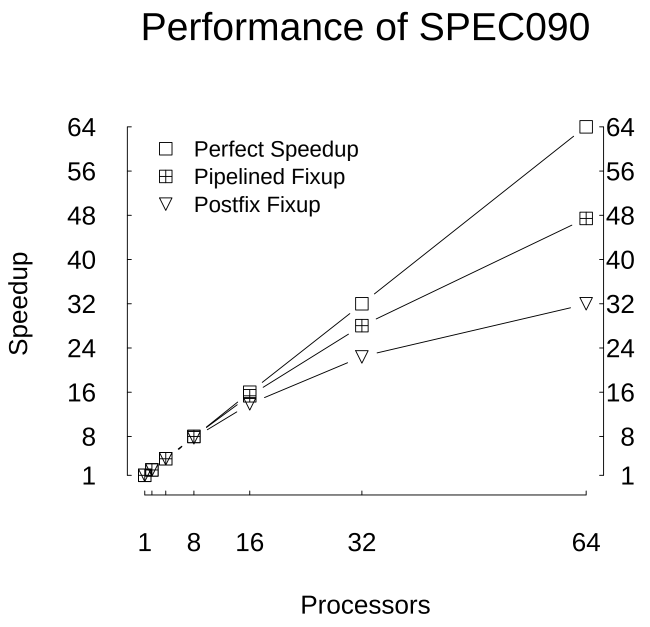

Figure 6: Speedups on `spec090`, for both pipelined and postfix fix-up

	Sparc-20	Paragon Nodes				Level-by-Level:C					Geometric
Trace		1	4	16	64	4	16	64	256	1024	
spec001	35.2	59.4	14.7	3.9	1.23	0.13	0.58	2.4	9.6	38.6	22.5
spec026	31.8	70.8	17.6	4.46	1.29	0.13	0.63	2.7	10.9	43.9	22.5
spec090	33.6	61.7	15.0	4.0	1.3	0.13	0.62	2.6	10.7	43.1	22.5

Table 5: Wallclock processing time per reference, in microseconds, for various parallel implementations on three SPEC benchmark traces.

and the one-processor version on a Sparc-20. All SIMD runs were performed on a 1K PE MP-2, on the first 2^{20} references of the trace. One first notes that the processing time per reference does vary from trace to trace. On the Paragon this is due to differences in average stack distance, as the underlying serial algorithm's cost increases with increasing average stack distance. This may explain why the level-by-level results begin to differ for large values of C. As noted before, our implementation of the general geometric algorithm is completely insensitive to the trace.

In comparing these numbers, we observe that our implementation of the general geometric algorithm is simply not competitive. This highlights the previously expressed need to optimize the $n(t)$ calculation. If the geometric algorithm were, say, 10 times faster, it would enough faster than even the Sparc-20 implementation to consider (provided one had sufficient IO capability to deal with much longer traces adequately). Next we observe that the level-by-level algorithm is extremely competitive—on small cache sizes. Fortunately small set sizes are the norm in studies of computer caches; this algorithm promises very high performance in that setting. But for the problem of computing all LRU stack distances regardless of cache size, one must conclude that the most robust approach considered here is MIMD. The size of the MIMD processor memories and the very low coupling between processors throughout the life of the computation promise excellent performance gains. Furthermore, parallelization requires only relatively minor modifications to existing efficient serial simulators.

6 Conclusions

This paper studies the performance of five algorithms for trace-driven simulation of a fully associative cache that use the LRU replacement policy. Three of the algorithms are SIMD, and are implemented on the MasPar MP-2. Performance data from 1K, 4K and 16K PE machines is presented. Two of the algorithms are MIMD, and are implemented on the Intel Paragon. Performance data using 1 to 64 processors is reported. We study the implementation's sensitivity to various parameters using randomly generated traces; we also report performance achieved using 8 to 95 million reference traces of several SPEC benchmark programs.

We find that an SIMD algorithm that presumes small limits on set size is fastest, but that the MIMD algorithms are the most robust. We also discover that on the traces considered, the scalable version of the MIMD algorithm has sometimes markedly poorer performance than its non-scalable counterpart. However, the relative different between pipelined and associative fixup schemes is likely to be different under the realistic assumption of many sets (in a set-associative cache) and a small upper bound on the number of lines permitted each set.

Cache simulation is an important part of designing many different types of computer subsystems. Trace-driven cache simulations are computationally intensive, but parallelizable. The experiments reported in this paper help to clarify the issues involved in choosing an approach for parallelized trace-driven cache simulation.

Acknowledgements

We gratefully acknowledge the help of Tim Busse (MasPar Corporation) who ran our codes on 4K and 16K PE MP-2s owned by MasPar Corporation. We also acknowledge the help of Jan Prins (Univ. of North Carolina) for providing MasPar source code for bitonic sorting algorithms. We also thank the University of New Mexico for access to SPEC benchmark traces made available through their TraceBase facility, and thank Mark Hill for directing us to that facility.

References

[1] J. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23:214–219, 1980.

[2] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[3] P. Heidelberger and H. Stone. Parallel trace-driven cache simulation by time partitioning. In *Proceedings of the 1990 Winter Simulation Conference*, pages 734–737, 1990.

[4] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin Cummings, Redwood City, CA, 1994.

[5] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 12(2):78–117, 1970.

[6] D. Nicol, A. Greenberg, and B. Lubachevsky. Massively parallel algorithms for trace-driven cache simulations. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):849–859, August 1994.

[7] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawerence Berkeley Laboratory.

[8] J.F. Prins. Efficient bitonic sorting of large arrays on the maspar mp-1. In *Proceedings of the* 3^{rd} *Symposium on the Frontiers of Massively Parallel Computation*, 1990.

[9] System Performance Evaluation Cooperative. *SPEC SDM Release 1.0 Technical Fact Sheet*, 1991.

[10] J.G. Thompson. *Efficient Analysis of Caching Systems.* PhD thesis, University of California, Berkeley, 1987.